PREPRINT UCRL- 81922

Lawrence Livermore Laboratory

SCIENTIFIC APPLICATION CODING IN THE CONTEXT OF DATAFLOW

John P. Woodruff

November 15, 1978

This paper was prepared for presentation at the National Computer Conference 1979, New York, New York, June 4-7, 1979.

SCIENTIFIC APPLICATION CODING IN THE CONTEXT OF DATAFLOW

John P. Woodruff
Lawrence Livermore Laboratory
Livermore, California

ABSTRACT

One potential use for dataflow computers is in scientific application computing: the numerical solution of partial differential equations. This article describes the type of computation that is carried out in support of applied physics research, with particular emphasis on the data dependencies that exist in both explicit and implicit finite difference equations. We describe the features of these application codes that determine the scale of the computation to be performed, and observe that present codes running on available computers are smaller scale solutions than ideal physical modeling would require. One possible avenue to significantly faster computation is the dataflow concept. Before programs can be written for a hypothetical dataflow computer, several programming techniques need to be developed. These techniques are embodied in the programming languages and the data types that are used to express a program. Imperative and applicative language styles are described and compared to the semantic basis of a dataflow computer. Data types that are used in conventional computer codes are examined, and certain shortcomings relative to dataflow computing are noted. We observe the need for a data type that can make the data dependencies between elements of arraylike entities explicit.

INTRODUCTION

Scientific computation in support of applied physics research is directed at the numerical solution of families of partial differential equations. Several disciplines in physics pursue the properties of sets of these equations. For example, continuum mechanics, heat conduction, and chemical kinetics can be expressed as sets of differential equations. In general, these families of equations express the space variation and time evolution of some measurable phenomena. The method of solution is to posit initial conditions and boundary conditions, and to iterate numerical approximations to the differential equations to compute a condition for a later time.

There is a well-established need to build codes that improve our physical modeling capability in three regards: dimensionality, spatial detail, and physical verisimilitude. All three demands drive us toward faster and larger computing machines. Experience shows that problems that are run tend to consume, at most, a few hours of computer time. When more powerful computers become available, more complicated codes are run, with the result that few

hour-size problems are more complicated, and represent more realistic physical models.

The dataflow computation paradigm has been proposed as a method for greatly increasing the number of numerical operations that can be performed in parallel during a code run. Codes that will exploit this paradigm need to be written in a way that exposes the inherent parallelism in the algorithm being solved. Novel language constructs and data types may be needed to facilitate the exposure of parallelism. This paper is an early progress report of work that is just beginning; some speculations of programming methods are presented, but we stress that only after a number of attempted applications have been tested will the field of dataflow programming be well defined.

DIFFERENTIAL EQUATIONS AND THEIR SOLUTIONS

An example of a partial differential equation of interest to a physicist is the Newtonian equation of motion, F = ma, expressed for a continuous isotropic medium:

$$\rho \frac{du}{dt} - \nabla S = 0 \qquad (1)$$

Here ρ is the density of the medium, u is the velocity field, and S is the stress field. The operator ∇ is the space difference operator, and d/dt is a time difference operator. This equation is not solvable by itself because it contains three independent variables. The complete formulation of shock hydrodynamics requires additional equations that are to be solved simultaneously for all the variables that appear. The full specification for shock hydrodynamics requires solution of six equations in six unknowns.[1]

To compute results, we discretize the space coordinates into small zones and then solve the system of equations repetitively for successive instants in time. The full set of finite difference equations is solved for a particular value of time t^n. Then a computation is made over all zones to find the maximum stable value of Δt that will allow the next cycle computation to proceed. The data dependence of the stability rule is that all zones must be solved for all dynamic quantities before the single value of Δt can be computed.

The equation of motion is transformed into finite difference form in one space dimension, yielding:

$$u_k^{n+1} = u_k^n + 2\Delta t \, (S_k^n - S_{k-1}^n) \Big/ \left[\rho_k^n (x_{k+1}^n - x_k^n) + \rho_{k-1}^n (x_k^n - x_{k-1}^n)\right] \qquad (2)$$

The subscript k indexes the discrete point in space; the superscript n identifies the cycle number of the iteration. We represent the u_k as an array of real numbers. At each cycle, we replace the array u^n by the newly computed array u^{n+1}. This equation is an example of an explicit finite difference

equation; it can be solved for all the k points at cycle n+1 using only information that was known from cycle n.

A more common expression of the same partial differential equation is the elaboration in two space dimensions. In this formulation there are two orthogonal components of velocity, ux and uy. Each component is defined at every zone in the space within the problem boundary. For numerical computation, each component might be represented as a doubly subscripted array of real numbers. The two equations would possess the same property as the equation in one dimension, namely all points can be calculated from previously known values.

Consider another differential equation, that of heat conduction:[2]

$$\frac{\partial \varepsilon}{\partial \theta}\frac{d\theta}{dt} = \frac{1}{\rho}\nabla \cdot \kappa \nabla \theta \qquad (3)$$

This equation expresses the time variation of a temperature θ in terms of the local gradient of temperature $\nabla\theta$, the heat conductivity κ of the medium, and $\partial\varepsilon/\partial\theta$, the specific heat at constant volume of the medium.

When we write this equation in finite difference form, we use an implicit differencing technique and obtain, for one space dimension:

$$\theta_k^{n+1} = \theta_k^n + \frac{\Delta t}{M_k \left(\frac{\partial \varepsilon}{\partial \theta}\right)_k}\left[\kappa_k(\theta_{k+1}^{n+1} - \theta_k^{n+1}) - \kappa_{k-1}(\theta_k^{n+1} - \theta_{k-1}^{n+1})\right] \qquad (4)$$

Zonal mass M_k is computed from the density and geometry, and $\partial\varepsilon/\partial\theta$ is obtained from a formulation of material properties. The temperature data are represented by a singly subscripted array of real numbers. Notice the dependence of θ_k^{n+1} on θ_{k+1}^{n+1}.

This tridiagonal set of linear equations is solved by back substitution: a set of recurrence relations for the coefficients is generated by traversing the grid from left to right, and then the temperatures are computed by traversing the equations from right to left. If the partial differential equation, Eq. (3), is to be solved in a multidimensional geometry, the equations are formulated in alternating direction implicit form,[3] then solved by a similar technique to the one-dimensional case.

SCALE OF COMPUTATION

The partial differential equations written above represent arbitrarily complicated continuous spatial variation of the physical phenomena that are described. In converting to finite difference form for numerical solution, simplifications and approximations are introduced that limit the detail that can be simulated. The scale of the computational work to be done is determined by the dimensionality of the equations, by the detail that is represented by the

discretization, and by the degree of physical verisimilitude that the equations represent.

By writing out separate equations for each orthogonal coordinate, it is possible to express the finite difference solutions in either one, two, or three space dimensions. Only a few problems have interesting solutions in one dimension. These problems are either spherically symmetric or are infinite planes. Such a problem is quite small; usually the total number of zones in a one-dimensional problem is of order 10^2. A few minutes of computer time suffices for the solution of most one-dimensional problems.

Most important application code systems are solutions for two degrees of freedom. Most problems are representations of physical bodies that are figures of revolution. Problems in two dimensions typically have of order 10^4 zones and require one to ten hours for solution.

A few codes are being written that deal with three degrees of freedom. Present computers are neither large enough nor fast enough to deal with fully general three-dimensional problems, which would require of order 10^6 zones. Since each zone of a code requires the computation and storage of between 6 and 30 different physical parameters, the widespread computation of three-dimensional problems awaits larger and faster machines.

The scale of physical detail that can be represented in a code is determined by the number of zones in the code's data space. The realism of a particular computer result is greatest if the zones in the problem are of comparable size, because this allows the numerical differences to resemble the ideal differentials most closely. Sometimes a physically small feature is important to the modeling of a considerably larger system. In such a case, distributing zones over the system so as to retain correct representation of space derivatives may demand many zones. A code runs for a duration that is $O(n^{d+1})$ where n represents the zone count and d is the dimensionality of the solution.

The final determinant to the scale of computation is the selection of the physical approximation embodied in the partial differential equations being solved. In the example of the Newtonian equation of motion shown in Eq. (1), the stress field was the scalar field S. In shock hydrodynamics the stress S is given by $S = (-P - Q)$, where P is the hydrostatic pressure and Q is the artificial viscosity. Pressure is the physical response of real materials to compression and heating. The equation-of-state function that produces P in a hydrocode may be quite complicated; typically about one third of the machine time used by such a code is spent evaluating the equation of state. The artificial viscosity Q is a computational strategem employed to allow simulation of physical systems that may contain discontinuities.

There are physical situations in which the hydrodynamic stress is not an adequate representation of reality.[4] One such is the computation of the response of an elastic solid medium, such as the body of the earth, to transient accelerations. To calculate elastic response, we add terms to the stress field that represent the elastic body forces. In two dimensions, this stress field

has three additional terms, each of which is computed by auxiliary equations, and each of which introduces further variables to the formulation.

In summary, the calculations we perform now are not the most desirable solution to a set of perfectly represented physical principles. Instead, our present calculations represent a set of compromises to the size and speed of available computing machinery. There is not now any apparent limit to the demand for faster and larger computations.

PROGRAMS FOR DATAFLOW COMPUTERS

The concept of a dataflow computer has been described by several authors.[5-8] This concept is sharply different from conventional computing ideas, and one might expect that the programs written for dataflow machines might be sharply different from programs intended for conventional machines. This section points out the salient features of the dataflow concept as seen by a programmer about to write a code for a dataflow machine, and indicates directions that programming research may take in developing methods for applying the dataflow paradigm to scientific computation.

A program on a conventional computer is a sequence of imperative instructions. In writing such a code, a programmer describes a set of activities; the machine executes these activities in prescribed order. The concept of control flow underlies the programmer's activities when constructing the procedural description of a code. In dataflow, control flow is not a useful paradigm. An instruction in dataflow does not execute because control comes to the instruction, but instead an instruction executes when the required input data have arrived. Thus the programmer of a dataflow machine has little use for a procedural description of the code's actions. It now appears that the procedural coding paradigm will be replaced by the idea of functionality of expressions. Programs on dataflow machines evaluate expressions instead of following sequential instructions.

A second major feature of conventional machines that is absent in dataflow machines is the random access memory that stores and retrieves values when keyed by an address. Data in a conventional computer have a kind of dual existence: as values and as addresses. The data types that have been invented for programming languages are influenced by this duality. An extreme example of the identification between program abstraction and physical memory in a computer is the vector: programming a procedure for a vector machine requires the programmer to force the abstract data (the values to be manipulated) into strict conformance with the physical layout of the machine. By contrast, the dataflow machine does not invite the contemplation of the dual value/address properties of data. The values of data entities exist, and travel between operations, but are not referred to by address. The aspect of code data that is important to dataflow programs is the ordering of actions that are used to create and transform the data entities. Functions that pass data values between each other are sequenced by their data dependencies. Functions that do not share any

common data declarations are not in any stated sequence, and therefore are candidates for concurrent execution.

The removal of these two familiar precepts leaves a void to be filled by new programming techniques. The development of innovative programming languages is one research topic that is aiding the dataflow programmer. Another area of research that is just emerging is the specification of data types that clearly express dependencies between elements of data.

LANGUAGE ISSUES

The dataflow graphs that represent the low level picture of the computation in a dataflow machine are apparently not highly useful for expressing algorithms. A human-engineered language representation is appropriate. Language processors will compile this high level code to dataflow graphs. Two language types have been advanced for this purpose.

Conventional languages such as Fortran and Pascal that are used for numerical computation are imperative languages. They are built to express procedural descriptions of programs. These languages are well suited to the semantics of conventional computers. Sequential control is reflected by language constructs such as conditional statements and looping structures; the concept of memory addressing is reflected in the ability of the language to express arbitrary assignments of values to identifiers. Such a language does not inherently make obvious the data dependencies between operators, so one can not always easily detect from the structure of a code which data elements are input to or output from a particular module. Still, there is a large amount of code already written in these languages. It would be very worthwhile to be able to run these existing codes on our hypothetical dataflow computers.

A better match to the semantics of the dataflow computer is to be found in the semantics of applicative languages. Here the single assignment rule is in force: only one definition of the value of any identifier is allowed. The identifier (variable name) is not seen as a place in memory holding a value, but as a label for a piece of information that is passed from some expression that created its value to some other operators that will use that value. In writing a code in an applicative language we write functions to be evaluated. The functions produce results that depend only on the values that were input to them. Further, functions may have no effect on any program data except their own output values, so side effects are not possible. These rules closely parallel the semantics of the dataflow paradigm. It takes a little practice for a programmer to switch from the imperative to the applicative style, but the emergence of human-engineered applicative languages with high expressive power for numerical applications will ease this transition.

DATA TYPES

In the foregoing discussion of application codes, the array data type figured prominently as the data abstraction that unified the concept of physical field and allowed the programmer to access the individual components of the field. Conventional machine semantics of undisciplined sequential fetches and stores of data is consistent with the array concept of contiguous physical memory. In dataflow, however, we need to express the data dependencies of individual results more clearly. The array in dataflow semantics is too large an assemblage of values to allow all possible parallelism in the algorithm that creates the values to show. An illustration of the possible interpretation of data dependencies in an application shows the weakness of the array. From Eq. (1), the new value of the velocity array in a Newtonian hydrocode is functionally

$$u = f_1\ (\rho, S, x, \Delta t) \tag{5}$$

This statement declares that the array u is to be delivered as a result of calculations using the arrays x, ρ, and S. When this function is computed on a dataflow machine, no subsequent use of any elements of array u can be made until the array is complete. This is because in general any element of an array may be data dependent on any of the input argument elements, and therefore it is not safe to extract results from u until it is certain that all have been defined.

The algorithm (Eq. (2)) that we use to compute u shows the actual functionality of the elements of u to be defined more tightly than the foregoing array functionality suggested:

$$u_k = f_2\ (\rho_{k-1,k}, S_{k-1,k}, x_{k-1,k,k+1}, \Delta t) \tag{6}$$

This functionality allows some elements of the array u to be created when only a specific few elements of the input arrays are in hand. If it were possible for the u data to be constructed piece by piece, then consumption of some elements of u could begin before all the elements were built. Such overlap might expose considerably more run-time parallelism than is available using array semantics.

The stream[9,10] appears to fulfill the need for an asynchronous data type to replace one-dimensional arrays. The stream could be used to carry part of the output from the function that creates u to the next function so that the successor could begin consuming values before all the elements of the structure were built. A possible shortcoming of the stream type is that the elements of the stream would be emitted in order, even though there is no inherent reason in the algorithm for this sequencing of elements. Unnecessary delays in consumption of stream elements might arise if differences in timing existed for different element creations.

A more difficult problem arises when we consider two-dimensional data structures. The functionality of a function that creates elements for a rectangular array in a two-dimensional hydrocode might be

$$ux_{k;\ell} = f_3\ (\rho_{k-1,k;\ell-1,\ell}, S_{k-1,k;\ell-1,\ell}, x_{k-1,k,k+1;\ell-1,\ell,\ell+1}, \Delta t) \qquad (7)$$

It is plain that the production of components of ux depend only locally on the elements of the input argument structures. There is no data type currently defined that allows this locality of data reference to be expressed, and it seems likely that much of the parallelism that could be exploited between disjoint data localities will be unavailable in the absence of such a data type. Possibly some generalization of a streamlike entity should be invented to allow these facts about data dependencies to be expressed.

SIMULATION

Our study of the application of dataflow semantics to the solution of scientific computations uses an experimental approach. We write code fragments and run them on software simulators[11,12] of dataflow machines to discover how well parallelism is exploited by a particular problem statement. This experimental approach has already led to the observation that data types stronger than array are needed; however no simulation using a streamlike type has yet been performed. We expect to test other hypotheses and to sharpen our dataflow programming skill using simulators. We will be reporting from time to time on the utility of the dataflow paradigm for scientific computing.

ACKNOWLEDGMENTS

The work reported here was performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore Laboratory under contract No. W-7405-Eng-48. This work was supported in part by the U.S. Department of Energy, Office of Basic Energy Sciences.

REFERENCES

1. R.D. Richtmyer and K.W. Morton, *Difference Methods for Initial-Value Problems* (Interscience Publishers, New York, 1967), 2nd ed., Chapter 12.

2. ibid, Chapter 8.

3. R.S. Varga, *Matrix Iterative Analysis* (Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1962), Chapter 7.

4. M.L. Wilkins, *Calculation of Elastic-Plastic Flow*, Lawrence Livermore Laboratory, Livermore, Calif., UCRL-7322, Rev 1 (1969).

5. J.B. Dennis, D.P. Misunas, and C.K. Leung, *A Highly Parallel Processor Using Data Flow Language*, Massachusetts Institute of Technology, Cambridge, Mass., Computation Structures Group Memo 134, (1977).

6. Arvind and K.P. Gostelow,"A Computer Capable of Exchanging Processors for Time," in *Information Processing 77*, B. Gilchrist, Ed. (IFIP, North Holland Publishing Company, 1977), pp. 849-853.

7. A.L. Davis, "Principles for Distributed Control Computer Architecture," in *Proc. 2nd Rocky Mountain Symposium on Microcomputers*, Pingree Park, Colorado, August 1978, (Institute of Electrical and Electronics Engineers, Inc., New York, 1978), pp. 108-123.

8. S. Patil, R.M. Keller, and G. Lindstrom, *An Architecture for a Loosely Coupled Parallel Processor*, Department of Computer Science, University of Utah, Salt Lake City, Utah, UUCS-78-105 (1978).

9. K.S. Weng, *Stream-Oriented Computation in Recursive Data Flow Schemes*, Massachusetts Institute of Technology, Cambridge, Mass., MAC Technical Memorandum 68, (1975).

10. Arvind, K.P. Gostelow, and W. Plouffe, *The (Preliminary) Id Report: An Asynchronous Programming Language and Computing Machine*, Department of Information and Computer Science, University of California, Irvine, Calif., Technical Report 114, (1978), Chapter 4.

11. K.P. Gostelow, University of California, Irvine, private communication, (September, 1978).

12. A.E. Oldehoeft, S. Allan, S. Thoreson, C. Retnadhas, and R. Zingg, *Translation of High Level Programs to Data Flow and Their Simulated Execution on a Feedback Interpreter*, Department of Computer Science, Iowa State University, Ames, Iowa, Technical Report # 78-2, (1978).

NOTICE

"This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Department of Energy, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately-owned rights."

NOTICE

Reference to a company or product name does not imply approval or recommendation of the product by the University of California or the U.S. Department of Energy to the exclusion of others that may be suitable.

JJP/jw/crs